PROLOG

Bernie Garrett
University of British Columbia, Canada

INTRODUCTION

Prolog is a logic based programming language, and was developed in the early 1970s and is a practical programming language particularly useful for knowledge representation and artificial intelligence (AI) applications. Prolog is different from many common computer languages in that it is not a procedural language (such as Basic, C, or Java). It is an interpreted logic based declarative language and as such has no loops, jumps, type declarations or arrays, and no fixed control constructs. In the past this has led to the impression that Prolog is a restricted language, useful only for highly specialized programming tasks by enthusiasts (Callear, 1994; Krzysztof, 1997). However, this is not the case and modern versions of Prolog are well equipped and versatile, and can be used for any programming task. The latest generations of the language (e.g., Visual Prolog) can also be integrated into more common object oriented languages.

BACKGROUND

Origins

The development and growth in the use of prolog has followed the expansion of interest in artificial intelligence and knowledge based/expert systems. These are computer systems that simulate human cognitive processes, and incorporate large volumes of information in a database using rules to attempt to encapsulate this information as knowledge (or the knowledge of a human expert in the case of expert systems).

Prolog was developed by Alain Colmerauer of Marseilles University, and Robert Kowalski of the University of Edinburgh, in the early 1970s as an alternative to the American Lisp programming languages (early mathematical notation based languages), and Planner (a procedural language representing "knowledge" in the form of high level procedural plans). Kowalski, was a primary advocate in the logic paradigm community (see Fundamental Ideas), and in collaboration with Alain Colmerauer they created a subset of the language "micro planner" called Prolog, Kowalski hoped to demonstrate with Prolog that the logic paradigm was a viable approach to programming. It was Philippe Roussel (also at Marseilles University) who came up with the name as an abbreviation for "PROgrammation en LOGique" to refer to this software tool which was originally devised as a man-machine interface using natural language.

Fundamental Ideas

Prolog is a declarative language in that all the facts and data relating to the subject domain are stored and statically declared in a Prolog database. Rules are created that draw out the information from the database as necessary. Problem solving is achieved from the perspective of the data rather than the procedure, and this can be highly efficient (Bratko, 1996). We can contrast this with the conventional procedural paradigm where the computer performs a sequence of instructions or procedures to resolve a problem. Prolog does not specify any data types in its structure in the way common programming languages do. It therefore has a very open data structure and does not distinguish integers from real numbers, for example. Prolog has two basic functional components. Firstly, a query interpreter program that searches the second component, a Prolog database of facts and rules. The database or program is normally in the form of a text file.

The Logic Paradigm

John McCarthy (1958) originally proposed that mathematical logic be used for representing the nature of knowledge in computer systems. Marvin Minsky and Seymour Papert developed a different approach based on procedural implementations at MIT where the program simply contains a series of computational steps to be carried out to reach a goal (Hewitt, 2006). The logic programming paradigm developed as an alternative to the procedural paradigm and incorporates the invocation of procedures from inferential and deductive processes. Many people were involved in the endeavor of deriving a computer programming language from the discipline of logic, notably Robert Kowalski at Edinburgh University.

Unlike most procedural languages, Prolog programs are not written in a way that models how a computer works, but incorporate techniques that reflect the logical principals of problem solving. In Prolog rather than describing how to compute a solution, the program consists of a data base of facts (or defined predicates about something) and logical relationships (rules) which describe the relationships which hold between those facts. Rather then running a program based on a set of procedures to find a solution to a problem, the logic paradigm makes the user ask a question. A runtime system then searches through a database of facts and rules using logical deduction to determine the answer to this

question, and then invocates a predetermined procedure as a result.

In reality Prolog is not a full implementation of logic programming as this would be purely declarative. It is more accurate to say that Prolog is a programming language based on logic as its implementation has distinct procedural aspects, such as backtracking (Hewitt & Agha, 1988). However, this is a useful aspect of the language as it makes it straightforward to write conventional computer programs in Prolog.

Backward and Forward Chaining

There are two main methods of reasoning when using inference rules in computer applications. These are forward and backward chaining. Forward chaining starts with the available data and uses inference rules to extract more data until a goal is reached. In backward chaining, the system starts with a list of goals and works backwards to see if there are data available that will support any of these goals. An inference engine using backward chaining searches all the inference rules until it finds one which has a then clause that matches a desired goal. If the if clause of that rule is not known to be true, then it is added to a list of goals to be searched for and the search continues until all the goals are met (or fail to be met). Backward chaining attempts to match the action rather than the conditions during its operation, and works from goals to facts. It eliminates the need to solve every possible outcome for a given set of rules. Forward chaining inference is often called data driven in contrast to backward chaining inference, which is referred to as goal driven reasoning. Prolog is based on mathematical logic, and the basis for this claim is that Prolog uses backward chaining processes in its operation from goal to sub-goal.

This process can be represented by the following code in Prolog:

$goal_X$:- $subgoal_1$, ..., $subgoal_n$.

This states that in order to prove $goal_X$, then you must prove $subgoal_1$ through to $subgoal_n$.

Unification

Unification is the built-in pattern-matching algorithm in Prolog, and one of the main concepts in behind it (Sterling & Shapiro, 1994). It is the mechanism by which variables are bound (or instantiated) to unique assignments. In Prolog, this operation is denoted by the equal symbol (=).

Queries in Prolog work by pattern matching. The query pattern is the goal, and if a fact in the Prolog database matches this goal, then the query succeeds and Prolog responds with 'yes.' If there is no matching fact, then the query fails and Prolog responds with 'no.'

Example:
In the following example Prolog unifies the variable "what" with the atom (a constant string of characters) "trees."

```
?- climbs(bear,What).
What=trees.
Prolog responds: yes
```

Prolog uses a capital letter to indicate a variable and a lowercase letter to indicate an atom. In older versions of Prolog a variable which is has not been instantiated yet can be unified with any atom, term, or another uninstantiated variable. In more modern versions a variable cannot be unified with a term that contains it. Binding a variable to a structure containing that variable can results in a cyclic structure which would cause the unification to loop forever. For example, A=f(A). This is a type of recursion. Modern versions of Prolog include an "occurs check" to prevent this happening.

Backtracking
In Prolog backtracking is a process that allows it to work through all the sub-goals in a rule if one sub-goal fails. In this case Prolog does not give up immediately and make the rule fail but it backtracks to previous sub-goals to try other instances of them in the database, then move forward again and see whether this causes the failed sub-goal to succeed. In this way it goes through a process that tries all the possible combinations of solutions, and finds the successful ones, before it finally reports that a rule has failed (Coehlo & Cotta, 1988).

In order to cope with the very limited memory systems and sequential computer architecture that were available when the language was developed, an efficient backtracking control structure was implemented so that only one possible computational path had to be stored at a time. This backtracking process is a method that has to be used on a sequential computer, which can only do one thing at a time and has to work through all the possibilities systematically. As it searches, Prolog leaves markers at points in the database to which it returns if a path down a particular branch fails to yield a resulting match. This exhaustive search method used by Prolog is called a depth first search method (Nilsson & Maluszynski, 1995). Diagram 1 demonstrates how this works in practice. The tree shows possible solutions for a supervision rule. Prolog finds "major(lewis)" first and then explores all possibilities of the "corporal" predicate before moving on to "major(lee)." It then explores all the possibilities of "corporal" again going as deep down the branches as possible.